⇒ Heaps Data Structure
⇒ Heapsort Algorithm

Motivation Suppose we want to maintain a list with at most n numbers that supports:

- Insertion
- Deletion
- GetMin

Attempt 1 Array with pointer to minimum element.

- Insertion - O(1)
- Deletion - O(n) → since we may remove the minimum element and would need to find new minimum
- GetMin - O(1)

Attempt 2 Sorted list

- Insertion - O(n)
- Deletion - O(n)
- GetMin - O(1)

→ If the list is an array, finding position to insert takes O(logn) with binary search, but inserting the element takes O(n).

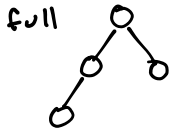
→ If the list is a linked list, then inserting is easy but binary search is not supported.

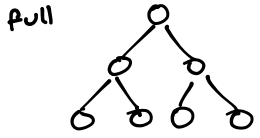
Heap Data Structure

- Insertion - O(logn)
- Deletion - O(log n)
- GetMin - O(1)

Definition A binary tree is full if the tree is completely filled in all levels except possibly in the lowest level

Examples:



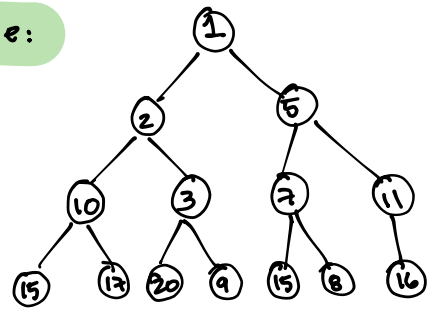


Heaps

- Conceptually we think of heap as a **full binary tree** that contains a number at each node.
- The numbers in the tree satisfy the **Min Heap Property**:

> The number at every node is greater than or equal to the number of the parent of the node.

Example:

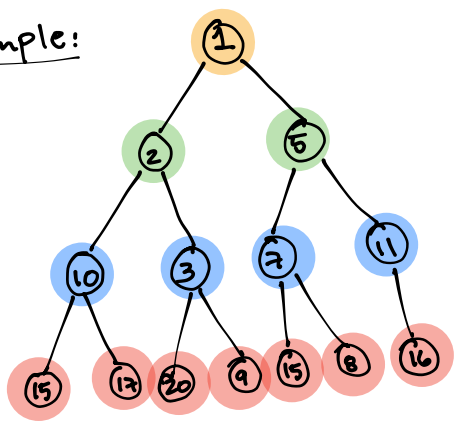


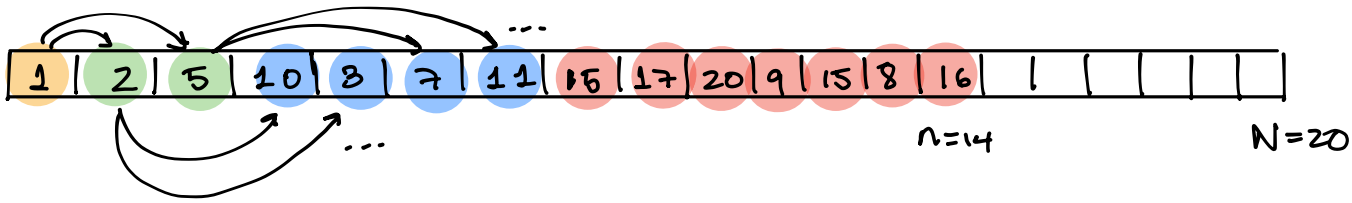
How to store heaps in memory?

Option 1: Using pointers, but arrays are faster.

- N = maximum capacity of the heap.
- Create array H with N elements.
- We keep track of the number of elements in the heap with a second variable n
- Each heap node corresponds to a position in H
- $H[0]$ contains the root
- The left child of $H[i]$ is at position $2i+1$ and the right child is at position $2(i+1)$

Example:





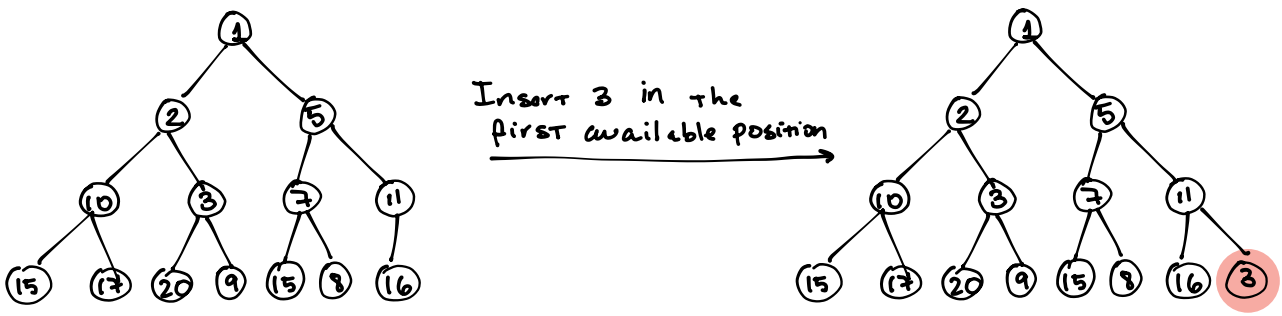
Note: The parent of $H[i]$ is at position $\lfloor \frac{i-1}{2} \rfloor$

Implementing Heap Operations

GetMin Takes $O(1)$, all we have to do is return $H[0]$.

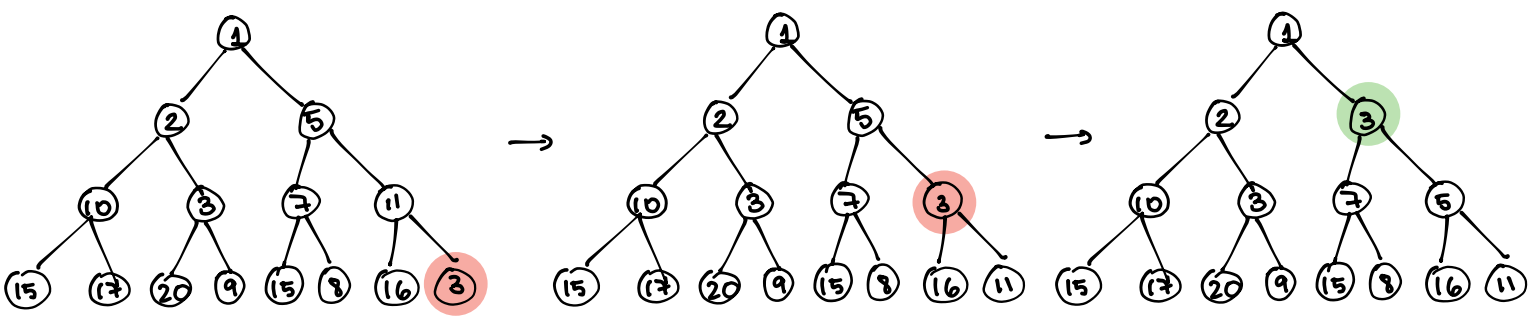
Insertion

Example: Suppose we want to insert 3 in:



The node containing 3 does not satisfy the Min Heap Property.

Idea Let 3 "bubble up" by swapping until it finds the right position.



This process is called **Heapify-Up**

Exercise: Write code for Heapify-Up.

Insert

Input Heap H and number v

```
if n == N
    No capacity left; output error
H[n] = v
Heapify-Up (H, n)
n = n + 1.
```

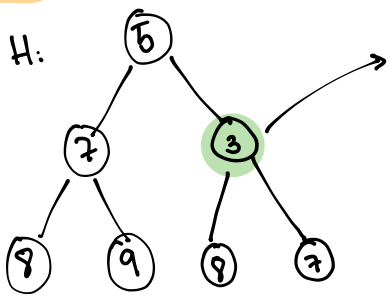
Correctness and Running Time of Insert

Definition: H is almost a heap with $H[i]$ too small if there is a value $\alpha \geq H[i]$ such that raising the value of $H[i]$ to α would make H a heap.

In words: All nodes but one satisfy the Min Heap Property and it is possible to increase the number in the incorrect node to make H a heap.

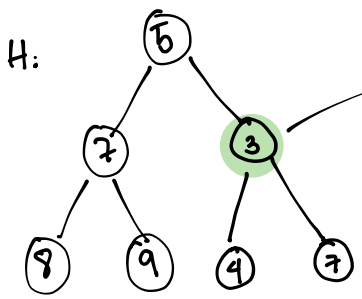
Example:

H:



Raising 3 to 6 would make H a heap, so H is almost a heap with $H[2]$ too small.

H:



3 can be raised up to 4, but this is not enough to make H a heap, so H is not almost a heap.

Fact 1: After inserting v to the end of the array, H is either a heap or almost a heap with $H[n]$ too small.

Lemma 2 If H is almost a heap with $H[i]$ too small, then the procedure Heapify-Up(H,i) makes H a heap in $O(\log n)$ time.

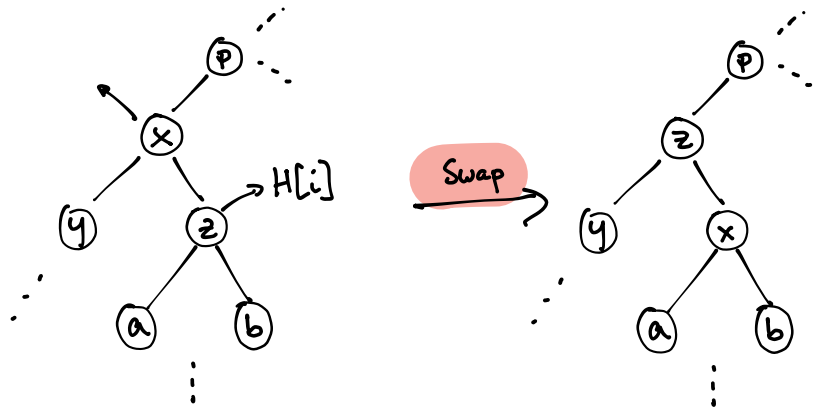
Proof

Base case: $i=0$

If H is almost a heap with $H[0]$ too small, then H is already a heap. This is because if raising $H[0]$ makes H a heap, then it must already be a heap.

Suppose $i \geq 1$ and that H is almost a heap with $H[i]$ too small but not a heap. Then, $H[i]$ and $H[\lfloor \frac{i}{2} \rfloor]$ are swapped.

We claim that after the swapping H is either a heap or almost a heap with $H[\lfloor \frac{i}{2} \rfloor]$ too small.



Hence, after swapping:

1) $x > z$ — Since for the swapping to occur, we must have: $H[\lfloor \frac{i}{2} \rfloor] = x > z = H[i]$. This implies that x satisfies the Min Heap Property.

2) $z < x \leq y$ — Before the swap node y satisfied the Min Heap Property, so $y \geq x$. This implies that node y satisfies the Min Heap Property after the swap.

3) $x \leq a$, $x \leq b$ — Since H is almost a heap with $x = H[i]$ too small, $\exists\, \alpha \geq z$ such that $\alpha \leq a$, $\alpha \leq b$ and $\alpha \geq x$. Therefore $a \geq \alpha \geq x$ and $b \geq \alpha \geq x$, so nodes a and b satisfy the Min Heap Property.

4) Finally, if z satisfies the Min Heap Property, H is a heap and we are done. Otherwise $p > z$, and if we increased z to p, then z would satisfy the Min Heap Property. Since $p \geq x$ and $p \geq y$, so would x and y.

The last observation is that the height of a heap with n elements is $O(\log n)$, so the number of swaps is at most $O(\log n)$. ∎

Remarks:

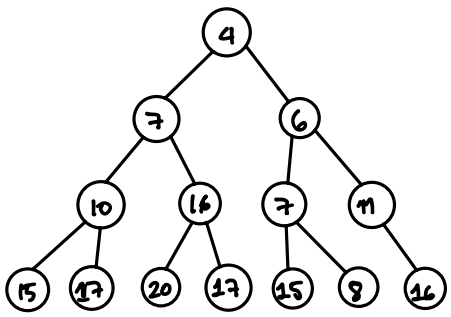
1. The heaps we have defined are called **Binary Min Heaps**. In some applications it is more convenient to use **d-ary heaps**. In addition, we can define **Max Heaps** in a similar manner.

2. Instead of storing numbers at each node, we could store [key, value] pairs. Then, the keys need to satisfy the Min or Max Heap Property.

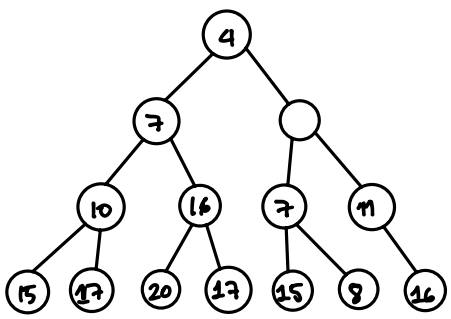
Deletion

⇒ Delete element at position i.

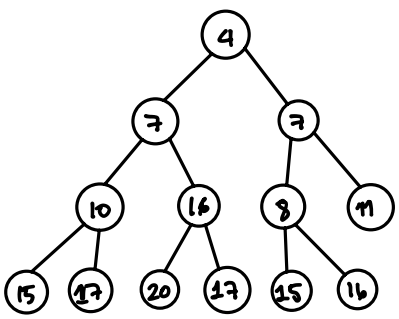
Example:



Delete position 2 ⟹



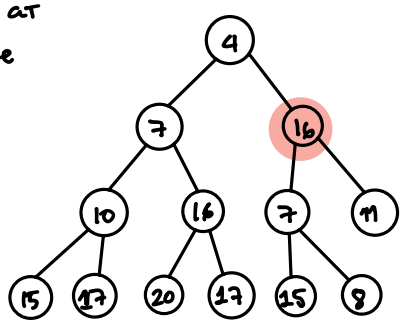
⇓ Move last element into empty position



The new element at position 2 may be too large or too small,

⇐

So it is pushed up or down with swaps if needed.



The process of pushing a node down with swaps is called **Heapify-Down**

Exercise: Write code for Heapify-Down

Delete

Input Heap H and position i

$H[i] = H[n-1]$

$n = n-1$

If $i > 0$ and $H[i] < H\left[\left\lfloor \frac{i-1}{2} \right\rfloor\right]$

  Heapify-Up (H, i).

If $(2i+1 < n$ and $H[i] > H[2i+1])$ or $(2(i+1) < n$ and $H[i] > H[2(i+1)]$

  Heapify-Down (H, i).

Correctness: The proof is analogous to that of the correctness of Insertion and we choose to omit it.
(If interested, check the proof of (2.13) in Kleinberg Tardos book).

Running Time: At most $O(\log n)$ swaps, so $O(\log n)$.

Heap Sort

We can use a min heap for sorting as follows:

1) Insert every element into a min heap.

2) Write down the root $H[0]$, delete it and repeat.

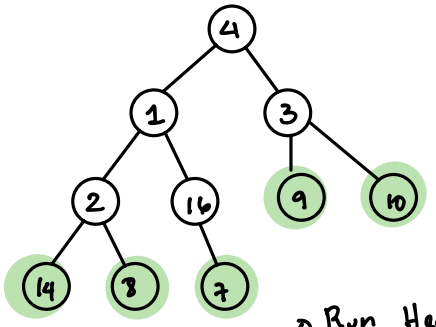
Running Time: $O(n \log n)$. Much faster than Insertion Sort!

However, this is not the best way of implementing HeapSort since it requires that we create a new array. That is, the sorting is not "in place" which is a highly coveted property for sorting algorithms, but we can implement heapsort "in place".

For this, we introduce the function Build-Heap that takes as input an array A and turns A into a heap.

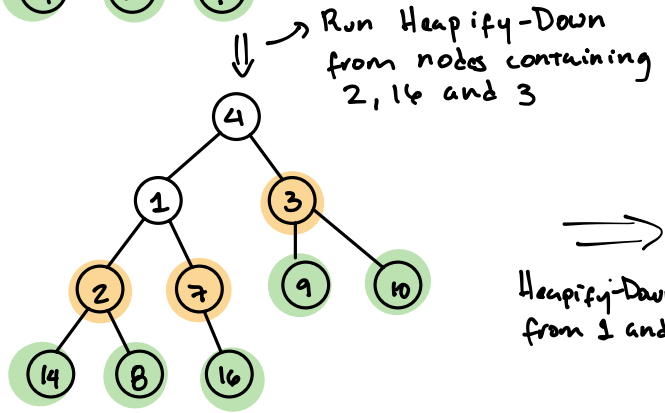
Example

A = | 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |



1) Many nodes violate the Min Heap Property.

Idea: Assume all leaves are Min heaps of one element. Then run Heapify-Down from all parents of leafs, then from their parents, and so on...



Fact The internal nodes of the heap occupy positions $0, 1, \ldots, \lfloor n/2 \rfloor - 1$ and the leafs positions $\lfloor n/2 \rfloor, \ldots, n-1$.

Build-Heap.

Input: an array A of n numbers.

for $i = \lfloor n/2 \rfloor - 1$ down to 0

Heapify-Down (A, i)

Running Time: $O(n \log n)$, since Heapify-Down is called at most $O(n)$ times. (A more careful analysis yields $O(n)$!)

Correctness:

Loop invariant: At the start of each for loop each node $i+1, \ldots, n$ is the root of a min heap

Proof. This is a true at the start since all leafs are min heaps, and it is maintained after each call to Heapify-Down. (Ex: work out details) ∎

The loop invariant is true at termination so $i=0$ is the root of a min heap and we are done.

With this new tools, we can now implement **heapsort** in place as follows.

Heapsort

Input : Array of n elements
Output : A sorted.

```
Build-Heap (A).

i = n-1

while i > 0
    x = H[0]
    H[0] = H[i]
    H[i] = x
    i = i - 1
    n = n - 1                    // n maintains capacity of heap
    Heapify-Down (H, 0)

Output Reverse of H.             (to avoid reverting use Max Heap instead)
```

Running Time : $O(n \log n)$

Correctness : Follows from the correctness established for the heap operations

Can we do better? No! Any sorting algorithm (that is based on comparisons) may require $\Omega(n \log n)$ comparisons

Rough Idea : There are $n!$ possible orderings an each comparison "got rids" of at most half of them. So, overall we need $\Omega(\log_2(n!)) = \Omega(n \log n)$ comparisons!

Note: We can do better if we assume something about input.